Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures

Angshuman Parashar[†] Michael Pellauer[†]
Michael Adler[†] Bushra Ahsan[†] Neal Crago[†] Daniel Lustig[⋆] Vladimir Pavlov[▷] Antonia Zhai[†◇]
Mohit Gambhir[†] Aamer Jaleel[†] Randy Allmon[†] Rachid Rayess[▷] Stephen Maresh[▷]
Joel Emer[†‡]

[†] VSSAD, [▷] Intel Corporation
Hudson, MA 01749

[‡] CSAIL, MIT
Cambridge, MA 02139

[⋆] Princeton University
Princeton, NJ 08544

[◇] University of Minnesota
Minneapolis, MN 55455

[†▷]{angshuman.parashar, michael.i.pellauer, michael.adler, bushra.ahsan, neal.c.crago, vladimir.pavlov, mohit.gambhir, aamer.jaleel, randy.allmon, rachid.e.rayess, stephen.maresh, joel.emer}@intel.com
[⋆]dlustig@princeton.edu [◇]zhai@cs.umn.edu [‡]emer@csail.mit.edu

ABSTRACT

In this paper, we present *triggered instructions*, a novel control paradigm for arrays of *processing elements* (PEs) aimed at exploiting spatial parallelism. Triggered instructions completely eliminate the program counter and allow programs to transition concisely between states without explicit branch instructions. They also allow efficient reactivity to inter-PE communication traffic. The approach provides a unified mechanism to avoid *over-serialized* execution, essentially achieving the effect of techniques such as dynamic instruction reordering and multithreading, which each require distinct hardware mechanisms in a traditional sequential architecture.

Our analysis shows that a triggered-instruction based spatial accelerator can achieve 8× greater area-normalized performance than a traditional general-purpose processor. Further analysis shows that triggered control reduces the number of static and dynamic instructions in the critical paths by 62% and 64% respectively over a program-counter style spatial baseline, resulting in a speedup of 2.0×.

Categories and Subject Descriptors

C.1.3 [**Computer Systems Organization**]: Processor Architectures—*Other Architecture Styles*

Keywords

Spatial Programming, Reconfigurable Accelerators

1. INTRODUCTION

Recently, SIMD/SIMT accelerators such as GPGPUs have been shown to be effective as offload engines when paired with general-purpose CPUs. This results in a complementary approach where the CPU is responsible for running the operating system and irregular programs, and the accelerator executes inner loops of uniform data-parallel code.

Unfortunately, not every workload exhibits sufficiently uniform data parallelism to take advantage of the efficiencies of this pairing. There remain many important workloads whose best-known implementation involves asynchronous actors performing different tasks, while frequently communicating with neighboring actors. The computation and communication characteristics of these workloads cause them to map efficiently onto *spatially-programmed* architectures such as field-programmable gate arrays (FPGAs). Furthermore, a number of important workload domains exhibit such kernels, such as signal processing, media codecs, cryptography, compression, pattern matching and sorting. As such, one way to boost the performance efficiency of these workloads is to add a new spatially-programmed accelerator to the system, complementing the existing SIMD/SIMT accelerators.

While FPGAs are very general in their ability to map the compute, control and communication structure of a workload, their *lookup table* (LUT) based datapaths are deficient in compute density compared to a traditional microprocessor — much less a SIMD engine. Furthermore, FPGAs suffer from a low-level programming model inherited from logic prototyping that includes unacceptably long compilation times, no support for dynamic context-switching, and often inscrutable debugging features.

Tiled arrays of coarse-grained ALU-style datapaths are known to achieve higher compute density than FPGAs [20, 13, 18]. A number of prior works [4, 12, 23] have proposed spatial architectures with a network of ALU-based *processing elements* (PEs) onto which operations are scheduled in systolic or dataflow order, with limited or no autonomous control at the PE level. Other approaches incorporate autonomous control at each PE using a *program counter* (PC) [24, 27, 21]. Unfortunately, as we will show, PC sequencing of ALU operations introduces several inefficiencies when attempting to capture intra- and inter-ALU control patterns of a frequently-communicating spatially-programmed fabric.

In this paper, we present *triggered instructions*, a novel control paradigm for ALU-style datapaths for use in arrays of PEs aimed at exploiting spatial parallelism. Trig-

ISCA '13 Tel Aviv, Israel

gered instructions remove the program counter completely, instead allowing the processing element to concisely transition between states of one or more *finite-state machines* (FSMs) without executing instructions in the datapath to determine the next state. This also allows the PE to react quickly to incoming messages on communication channels. In addition, triggered instructions provide a unified mechanism to avoid *over-serialized* execution, essentially achieving the effect of techniques such as dynamic instruction reordering and multithreading, which each require distinct hardware mechanisms in a traditional sequential architecture.

We evaluate the triggered-instruction approach by simulating a spatially-programmed accelerator on a range of workloads. Our analysis for this set of workloads, which span a range of algorithm classes not known to exhibit extensive uniform data parallelism, shows that such an accelerator can achieve 8× greater area-normalized performance than a traditional general-purpose processor. We provide further analysis of both a set of common control idioms and the critical paths of the workload programs to illustrate how a triggered instruction architecture contributes to this performance gain.

2. BACKGROUND AND MOTIVATION

2.1 Spatial Programming Architectures

Spatial programming is a paradigm whereby an algorithm's dataflow graph is broken into regions, which are connected by producer-consumer relationships. Input data is then streamed through this pipelined graph. Ideally, the number of operations in each stage is kept small, as performance is usually determined by the *rate-limiting step*.

Just as vectorizable algorithms see large efficiency boosts when run on a vector engine, workloads that are naturally amenable to spatial programming can see significant boosts when run on an enabling architecture. A traditional processor would execute such programs serially over *time*, but this does not result in any noticeable efficiency gain, and may even be slower than other expressions of the algorithm. A shared-memory multicore can improve this by mapping different stages onto different cores, but the small number of cores available relative to the large number of stages in the dataflow graph means that each core must multiplex between several stages, so the rate-limiting step generally remains large.

In contrast, a typical spatial-programming architecture is a fabric of hundreds of small processing elements (PE) connected directly via an on-chip network. Given enough PEs, an algorithm may be taken to the extreme of mapping a single operation in the kernel's dataflow graph to each PE, resulting in a very fine-grained pipeline. This is the approach taken by a number of *reconfigurable* architectures.

FPGAs are the most successful spatially-programmed reconfigurable architecture in use today. FPGAs are designed to emulate a broad range of logic circuits because they are primarily targeted at ASIC prototyping and replacement. Consequently, they use very fine-grain reconfigurable elements such as *lookup tables* (LUTs) [6, 17]. The LUTs are chained into larger operations using flexible-but-expensive on-chip networks. This generality limits the clock speed at which mapped designs can be run.

FPGAs also suffer from a low-level programming model due to its roots in logic prototyping. The generality and fine-granularity of LUTs and the interconnection network creates a large search space of solutions for place and route algorithms, leading to unacceptably long compilation times. Reprogramming an FPGA is also a slow process and is at odds with the rapid context-switches that a reconfigurable logic accelerator would be expected to support.

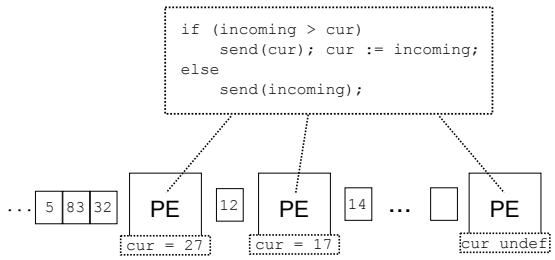


Figure 1: Example of a spatially-programmed sort.

When using reconfigurable architectures for direct algorithmic acceleration instead of logic prototyping, these issues can be partially addressed by the observation that the class of operations that the reconfigurable architecture needs to cover is more limited—particularly when used in conjunction with a traditional CPU. As observed by several efforts [20, 13, 18], this limited class of operations creates opportunities to achieve higher area density and better power/performance efficiency than conventional FPGAs while retaining sufficient flexibility. This has led to several proposals [4, 21, 24, 23, 20, 13, 18] that use an array of coarser-grained multi-bit ALUs as the datapath of PEs in a spatially-programmed architecture.

Within the domain of array-of-ALU approaches is a class of architectures that do not feature any autonomous control mechanism inside each ALU. These architectures are either purely systolic [16], statically map only one operation per ALU [12], or schedule operations onto the ALUs in strict dataflow order [4]. These architectures rely on being able to transform control flow graphs into predicated dataflow graphs. Such approaches are effective at mapping the control structures of a subset of problems, but do not approach the flexibility or generality of architectures with internal autonomous control at each PE. Another class of proposals calls for general autonomously-controlled PEs [24, 27, 21] using variants of the existing PC-based control model. The PC-based control model has historically been the best choice for standalone CPUs that run arbitrary and irregular programs. In the remainder of this section, we demonstrate that PC-based control introduces unacceptable inefficiencies in the context of spatial programming.

2.2 Spatial Programming Example

As a concrete example, let us explore how a well-known workload can benefit from spatial programming. Consider the simple spatially-mapped sorting program shown in Figure 1. In this approach, the worker PEs communicate in a straight pipeline. The unsorted array is streamed in by the first PE. Each PE simply compares the incoming element to the largest element seen so far. The larger of the two values is kept, and the smaller sent on. Thus after processing k elements worker 0 will be holding the largest element, and worker $k-1$ the smallest. The sorted result can then be streamed out to memory through the same straightline communication network.

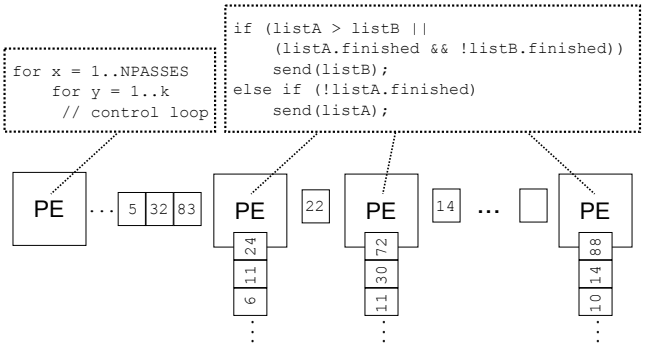


Figure 2: A more realistic spatial merge sort.

This example represents a limited toy workload in many ways—it requires k PEs to sort an array of size k, and worker 0 will do $k-1$ comparisons while worker $k-1$ will only do 1 (an insertion sort, with a total of k^2 comparisons). However, despite its naivete this workload demonstrates some remarkable properties. First, the peak utilization of the system is quite good—in the final step all k datapaths can simultaneously execute a comparison. Second, the communication between PEs is local and parallel—on a typical mesh fabric it is easy to map this workload so that no network contention will ever occur. Finally—and most interestingly—this approach sorts an array of size k with exactly k loads and k stores. The loads and stores that a traditional CPU must use to overcome its relatively small register file are replaced by direct PE-to-PE communication. This reduction in memory operations is critical in understanding the benefits of spatial programming. We have been able to characterize the benefits as follows:

- Direct communication uses roughly 20× lower power than communication through an L1 cache, as the overheads of tag matching, load-store queue search, and large data array read are removed.
- Cache coherence overheads, including network traffic and latency are likewise removed.
- Reduced memory traffic lowers cache pressure, which in turn increases effective memory bandwidth for remaining traffic.

Finally, it is straightforward to expand our toy example into a realistic merge sort engine able to sort a list of any size (Figure 2). First, we begin by programming a PE into a small control FSM that handles breaking the array into sub-arrays of size k and looping. Second, we slightly change the worker PEs' programming so that they are doing a merge of two distinct sorted sub-lists. With these changes our toy workload is now a radix k merge sort capable of sorting a list of size n in $n * log_k(n)$ loads. Because k can be in the hundreds for a reconfigurable fabric, the benefits can be quite large. In our experiments we observed 17× fewer memory operations compared to a general-purpose CPU and an area-normalized performance improvement of 8.8× (Section 5), which is better than the currently best-known GPGPU performance [19].

2.3 Limitations of PC-based Control

To illustrate the inefficiencies of program counters in the spatial programming context, let us code the merge sort PE shown in Figure 2. We must first address the representation of the queues that pass the sorted sub-lists between workers. In a multicore system, the typical approach is to use shared memory for the queue buffering, along with sophisticated polling mechanisms such as *memory monitors*. In a spatially-programmed fabric, having hundreds of PEs communicating using shared memory would create unacceptable bandwidth bottlenecks—in addition to increased overheads of pointer chasing, address offset arithmetic, and head/tail occupancy comparisons. Thus shared memory communication queues are not considered in this paper.

```
check_a: beqz   %in0.notEmpty, check_a // listA
check_b: beqz   %in1.notEmpty, check_b // listB
check_o: beqz   %out0.notFull, check_o // outList
         beq    %in0.tag, EOL, a_done
         beq    %in1.tag, EOL send_a
         cmp.lt %r0, %in0.first, %in1.first
         bnez   %r0, send_a
send_b:  enq    %out0, %in1.first
         deq    %in1
         jump   check_a
send_a:  enq    %out0, %in0.first
         deq    %in0
         jump   check_a
a_done:  beq    %in1.first, EOL, done
         jump   send_b
done:    deq    %in0
         deq    %in1
         return;
```

Static Insts	18
Avg Insts/Iteration	10
Avg Branches/Iteration	7

Figure 3: PC+RegQueue ISA merge sort worker representation using register-mapped queues.

Instead, let us assume that the ISA directly exposes data registers and status bits corresponding to direct communication channels between PEs. The ISA must contain a mechanism to query if the input channels are not empty, and output channels are not full, to read the first element, and to enqueue and dequeue. Furthermore we add an architecturally-visible tag to the channel that merge sort uses to indicate that the end of a sorted sub-list has been reached (`EOL`). A representation of the merge sort in this theoretical assembly language is given in Figure 3. Several inefficiencies are immediately noticeable. First, it uses *active polling* to test the queue status, an obvious power waste. Second, it falls victim to *over-serialization*. For example, if new data on `listA` arrives before that on `listB` there is no opportunity to begin processing the `listA`-specific part of the code. Finally, the code is quite branch-heavy when compared to that typically found on a traditional core, and some of these branches are hard to predict.

In order to be fair to this PC-based ISA we must try to improve the architecture somehow. Table 1 summarizes the techniques that we explore below.

One idea to improve queue accesses is to allow *destructive reads* of input channels. In such an ISA the SRC fields of the instruction are supplemented with a bit indicating whether a dequeue is desired. This is an important improvement because it reduces both static and dynamic instruction count. Merge sort's implementation on this architecture can remove 3 instructions compared to Figure 3.

The next idea is to replace the active polling with a

Feature	Description	Notes
PC (Baseline)	PEs use program counters, communicate using shared-memory queues.	High latency, bottlenecks.
+RegQueue	Expose register-mapped queues to ISA, test via active polling.	Poor power efficiency.
+FusedDeq	Destructive read of queue registers without separate instructions.	Good improvement.
+RegQSelect	Allow indirect jump based on register queue status bits.	Minimal improvement.
+RegQStall	Issue stalls on queue input/output registers without special instructions.	Bubbles, over-serialization.
+QMultiThread	Stalling on empty/full queue yields thread.	Significant additional hardware.
+Predication	Predicate registers that can be set using queue status bits.	Boolean expressions don't compose.
+Augmented	ISA augmented with all of the above features except +QMultiThread.	Used in our evaluations (Section 5).

Table 1: Adding features to a PC-based ISA to improve efficiency for spatial programming.

```
start:        beq    %in0.tag, EOL, a_done
              beq    %in1.tag, EOL, send_a
              cmp.ge p2, in0.first, in1.first
send_b:  (p2) enq    %out0, %in1.first (deq %in1)
send_a: (!p2) enq    %out0, in0.first (deq %in0)
              jump   start
a_done:       cmp.ne p2, %in1.first, EOL
         (p2) jump   send_b
              nop    (deq %in0, deq %in1)
              return;
```

Static Insts	9
Avg Insts/Iteration (Issued)	6
Avg Insts/Iteration (Committed)	5
Avg Branches/Iteration	3
Speedup vs PC+RegQueue (Fig 3)	1.4×

Figure 4: PC+Augmented ISA merge sort worker.

select—an indirect jump based on queue status bits. This is a marginal improvement in instruction count but does not help power efficiency. A better idea is to add *implicit stalling* to the ISA. In this case the queue registers such as `%in0` would be treated specially—any instruction that attempts to read/write them would require the issue logic to test the empty/full bits and delay issue until the status becomes correct. Merge sort's implementation on this architecture is the same as in Figure 3, but removes the first three instructions entirely.

Of course, the downside of this is that the ALU will not be used when the PE is stalled. Therefore the next logical extension is to consider a limited form of *multi-threading*. In this ISA any read/write of a queue would make the thread eligible to be switched out and replaced with a ready one. This is a promising approach, but we believe that the overheads associated with it—duplication of state resources, additional muxing, and scheduling fairness—run counter to the fundamental spatial-architecture principle of replicating simple PEs. In other words, the cost-to-benefit ratio of multithreading is unattractive. We reject *out-of-order issue* for similar reasons.

The final ISA extension we consider is *predication*. We define a variant of our ISA that is able to test and set a dedicated set of boolean predicate registers. Figure 4 shows a re-implementation of the merge sort worker in a language with predication, implicit stalling, and destructive reads. It is interesting to note how little predication improves the control flow of the example. This is because of several limitations:

- Instructions are unable to read multiple predicate registers at once (inefficient conjunction).
- Composing multiple predicates into more complex boolean expressions (disjunctions, etc) must be done using the ALU itself.
- Jumping between regions requires that the predicate expectations be set correctly. (Note that the branch from `a_finished` is forced to use `p2` with a positive polarity.)
- Predicated false instructions introduce bubbles into the pipeline (Section 4).

Taken together, these inefficiencies mean that conditional branching remains the most efficient way to express the majority of the code in Figure 4. While we could continue to try to add features to PC-based schemes in order to improve efficiency, in the remainder of the paper we demonstrate that taking a different approach altogether can efficiently address these issues while simultaneously removing over-serialization and providing the benefits of multi-threading.

3. TRIGGERED INSTRUCTIONS

A large degree of the inefficiency discussed in the previous section stems from the issue of efficiently composing boolean control flow decisions. In order to overcome this, we draw inspiration from the historical computing paradigm of *guarded actions*, a field that has a rich technical heritage including Dijkstra's language of guarded commands [8], Chandy and Misra's Unity [5], and the Bluespec hardware description language [3].

Computation in a traditional guarded action system is described using *rules* composed of *actions* — state transitions — and *guards* — boolean expressions that describe when a certain action is legal to apply. A *scheduler* is responsible for evaluating the guards of the actions in the system and posting *ready* actions for execution, taking into account both inter-action parallelism and available execution resources. Algorithm 1 illustrates our merge sort worker in traditional guarded action form. Note how this paradigm naturally separates the representation of data transformation (via actions) from the representation of control flow (via guards). Additionally, the inherent side-effect-free nature of the guards means that they are a good candidate for parallel evaluation by a hardware scheduler.

A *triggered instruction architecture* (TIA) applies this concept directly to controlling the scheduling of operations on a PE's datapath at an instruction-level granularity. In the historical guarded action programming paradigm, arbitrary boolean expressions are allowed in the guard, and arbitrary data transformations can be described in the action. To adapt this concept into an implementable ISA, both must be bounded in complexity. Furthermore, the scheduler must have the potential for efficient implementation in hardware. To this end, we define a limited set of operations and state updates that can be performed by the datapath (instructions) and a limited language of boolean expressions (triggers) built from a variety of possible queries on a PE's architectural state.

Algorithm 1 Traditional Guarded Action Merge Sort Worker

```
rule sendA
when listA.first() != EOL && listB.first() != EOL &&
listA.data < listB.data do
    outList.send(listA.first()); listA.deq();
end rule
rule sendB
when listA.first() != EOL && listB.first() != EOL &&
listA.data >= listB.data do
    outList.send(listB.first()); listB.deq();
end rule
rule drainA
when listA.first() != EOL && listB.first() == EOL do
    outList.send(listA.first()); listA.deq();
end rule
rule drainB
when listA.first() == EOL && listB.first() != EOL do
    outList.send(listB.first()); listB.deq();
end rule
rule bothDone
when listA.first() == EOL && listB.first() == EOL do
    listA.deq(); listB.deq();
end rule
```

The *architectural state* of our proposed TIA PE is composed of the following elements:

- A set of **data registers** (R/W).
- A set of **predicate registers** (R/W).
- A set of **input-channel head elements** (R-only).
- A set of **output-channel tail elements** (W-only).

Each channel has three components — *data*, a *tag* and a *status* predicate that reflects whether an input channel is empty or an output channel is full. Tags do not have any special semantic meaning — the programmer can use them in a variety of ways.

A *trigger* is a programmer-specified boolean expression formed from the logical conjunction[1] of a set of queries on the PE's architectural state. Triggers are evaluated by a hardware scheduler (described shortly). The set of allowable trigger query functions are carefully chosen to maintain scheduler efficiency while allowing for a large degree of generality in the useful expressions. These query functions are:

- **Predicate Register Values (optionally negated):** A trigger can specify a requirement for one or more predicate registers to be either true or false, e.g., `p0 && !p1 && p7`.
- **Input/Output Channel Status (implicit):** The scheduler implicitly adds the empty status bits for each operand input channel to the trigger for an instruction. Similarly, a not-full check is implicitly added to each output channel an instruction attempts to write. The programmer does not have to worry about these conditions, but must understand while writing code that the hardware will check them. This facilitates convenient, fine-grained, producer/consumer interaction.
- **Tag Comparisons against Input Channels:** A trigger may specify a value that an input channel's tag must match, e.g., `in0.tag == EOL`.

An *instruction* represents a set of data and predicate computations on operands drawn from the architectural state.

[1]Although the architecture natively allows only conjunctions in trigger expressions, disjunctions can be emulated by creating a separate triggered instruction for each disjunctive term.

```
doCheck:
    when (!p0 && %in0.tag != EOL
              && %in1.tag != EOL) do
        cmp.ge p1, %in0.data, %in1.data (p0 := 1)
sendA:
    when (p0 && p1) do
        enq %out0, %in0.data (deq %in0, p0 := 0)
sendB:
    when (p0 && !p1) do
        enq %out0, %in1.data (deq %in1, p0 := 0)
drainA:
    when (%in0.tag != EOL && %in1.tag == EOL) do
        enq %out0, %in0.data (deq %in0)
drainB:
    when (%in0.tag == EOL && %in1.tag != EOL) do
        enq %out0, %in1.data (deq %in1)
bothDone:
    when (%in0.tag == EOL && %in1.tag == EOL) do
        nop (deq %in0, deq %in1)
```

Static Insts	6
Avg Insts/Iteration	2
Speedup vs PC+RegQueue (Fig 3)	5×
Speedup vs PC+Augmented (Fig 4)	3×

Figure 5: Triggered instruction merge sort worker.

Instructions selected by the scheduler are executed on the PE's datapath. An instruction has the following read, compute and write capabilities:

- An instruction may **read** a number of operands, each of which can be data at the head of an input channel, a data register, or the vector of predicate registers.
- An instruction may **perform a data computation** using one of the standard functions provided by the datapath's ALU. It may also **generate one or more predicate values** that are either constants (true/false) or derived from the ALU result via a limited set of datapath-supported functions, e.g., reduction AND, OR and XOR operations, bit extractions, ALU flags such as overflow, etc.
- An instruction may **write** the data result and/or the derived predicate result into a set of destinations within the architectural state of the PE. Data results can be written into the tail of an output channel, a data register, or the vector of predicate registers. Predicate results can be written into one or more predicate registers.

Figure 5 shows our merge sort expressed using triggered instructions. Note the density of the trigger control decisions—each trigger reads at least two explicit boolean predicates. Additionally, conditions for the queues being *notEmpty* or *notFull* are recognized implicitly. Only the comparison between the actual multi-bit queue data values is done using the ALU datapath, as represented by the `doCheck` instruction. Predicate `p0` is used to indicate that the check has been performed, while `p1` holds the result of the comparison. Note also the lack of over-serialization. Only the explicitly programmer-managed sequencing using `p0` is present.

An example TIA PE is illustrated in Figure 6. The PE is pre-configured with a static set of instructions. The triggers for these instructions are then continuously evaluated by a

dedicated hardware scheduler that dispatches legal instructions to the datapath for execution. At any given scheduling step, the trigger for zero, one, or more instructions can evaluate to true. The guarded action model — and by extension our triggered instruction model — allows all such instructions to fire in parallel subject to datapath resource constraints and conflicts.

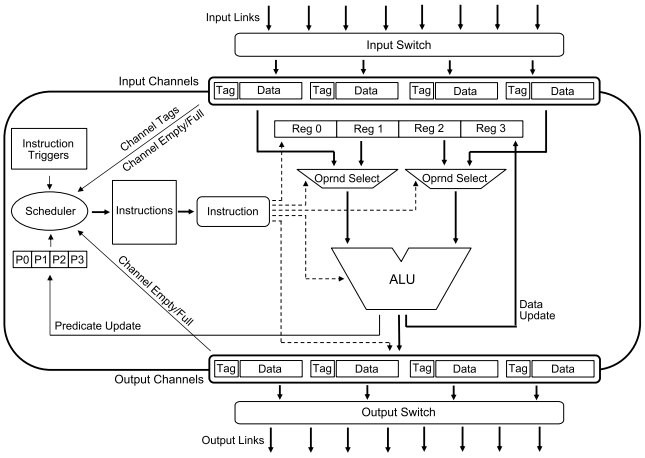


Figure 6: A triggered-instruction based PE.

The high-level microarchitecture of a TIA hardware scheduler is shown in Figure 7. The scheduler uses standard combinatorial logic to evaluate the programmer-specified query functions for each trigger based on values in the architectural state elements. This yields a set of instructions that are eligible for execution, among which the scheduler selects one or more depending on the datapath resources available. The example shown in this figure illustrates a scalar data-path that can only fire one instruction per cycle, therefore the scheduler selects one out of the available set of ready-to-fire instructions using a priority encoder.

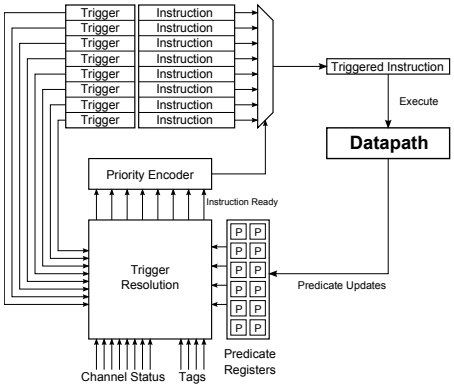


Figure 7: Microarchitecture of a TIA scheduler.

As with any architecture, a triggered-instruction architecture is subject to a number of parameterization options and their associated cost-vs-benefit tradeoffs. *Architectural* parameters include the number of instances of each class of architectural state element (data registers, predicate registers, etc.), the set of data and predicate functions supported by the datapath, the scope and flexibility of the trigger functions, and the number of input operands and output destinations. The design space of *microarchitectural* alternatives includes scheduler implementation choices, scalar vs. superscalar datapaths, pipelining strategies, etc. An exhaustive investigation of the entire design space is outside the scope of this work. To provide the reader with some intuition on what a reasonably balanced TIA PE could look like, we provide an example architectural configuration in Table 2. This is also the configuration we use for our evaluation in Section 5.

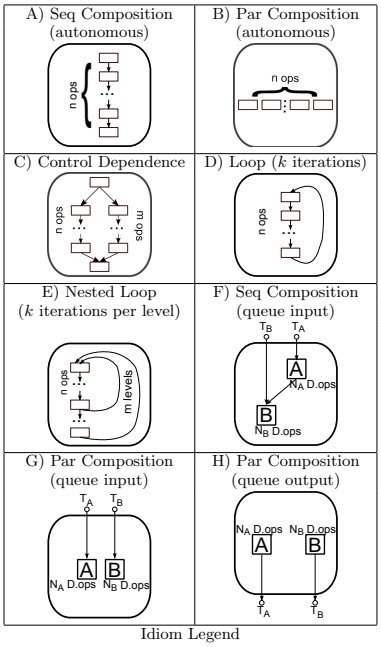
Sources per Instruction	2
Registers	8
Predicates	8
Max Triggered Instructions per PE	16

Table 2: Example PE Architecture Parameters.

3.1 Observations on the Triggered Model

Having defined the basic structure of a triggered instruction architecture, we are now in a position to make some key observations:

- A TIA PE does not have a program counter or any notion of a static *sequence* of instructions. Instead, there is a limited *pool* of triggered instructions that are constantly bidding for execution on the datapath. This fits very naturally into a spatial programming model where each PE is statically *configured* with a small pool of instructions instead of streaming in a sequence of instructions from an instruction cache.
- Observe that there are no branch or jump instructions in the triggered ISA—every instruction in the pool is eligible for execution if its trigger conditions are met. Thus, every triggered instruction can be viewed as a multi-way branch into a number of possible states in an FSM.
- With clever use of predicate registers, a TIA can be made to emulate the behavior of other control paradigms. For example, a sequential architecture can be emulated by setting up a vector of predicate registers to represent the current state in a sequence—essentially, a program counter. Predicate registers can also be used to emulate classic predication modes, branch delay slots and speculative execution. Triggered instructions is a superset of many traditional control paradigms. The cost of this generality is scheduler area and timing complexity, which imposes a restriction on the number of triggers (and thus, the number of instructions) that the hardware can monitor at all times. While this restriction would be crippling for a temporally programmed architecture, it is reasonable in a spatially-programmed framework because of the low number of instructions typically mapped to a pipeline stage in a spatial workload.
- The hardware scheduler is built from combinatorial logic — it simply is a tree of AND gates. This means that only the state equations that require re-evaluation will cause the corresponding wires in the scheduler logic to swing and consume dynamic power. In the absence of channel activity or internal state changes, the scheduler does not consume any dynamic power whatsoever. The same control equations would have been evaluated using a chain of branches in a PC-based architecture.



Idiom	PC+RegQueue	PC+Augmented	Triggered Instructions	TI Advantage over PC+Augmented
(A)	D.ops = n *(serialized)*	D.ops = n *(serialized)*	D.ops = n *(serialized)*	-
(B)	D.ops = n *(serialized)*	D.ops = n *(serialized)*	D.ops = n *(unordered)*	eliminates serialization
(C)	D.ops = m or $n+1^{\dagger}$ C.ops = 1 $\dagger$ *1 comparison*	D.ops = m or $n+1^{\dagger}$ F.ops = m or n $\dagger$ *1 comparison*	D.ops = m or $n+1^{\dagger}$ C.ops = 0; F.ops = 0 $\dagger$ *1 comparison*	eliminates m or n F.ops
(D)	D.ops = $n*k+k^{\dagger}$ C.ops = k $\dagger$ *k comparisons*	D.ops = $n*k+k^{\dagger}$ C.ops = k $\dagger$ *k comparisons*	D.ops = $n*k+k^{\dagger}$ C.ops = 0 $\dagger$ *k comparisons*	eliminates k C.ops
(E)	D.ops = k^m*n $+\frac{k(k^m-1)}{(k-1)}{}^{\dagger}$ C.ops = $\frac{k(k^m-1)}{(k-1)}$ $\dagger\frac{k(k^m-1)}{(k-1)}$ *comparisons*	D.ops = k^m*n $+\frac{k(k^m-1)}{(k-1)}{}^{\dagger}$ C.ops = $\frac{k(k^m-1)}{(k-1)}$ $\dagger\frac{k(k^m-1)}{(k-1)}$ *comparisons*	D.ops = k^m*n $+\frac{k(k^m-1)}{(k-1)}{}^{\dagger}$ C.ops = 0 $\dagger\frac{k(k^m-1)}{(k-1)}$ *comparisons*	eliminates $\frac{k(k^m-1)}{(k-1)}$ C.ops
(F)	D.ops = N_A+N_B Q.ops = 2	D.ops = N_A+N_B Q.ops = 0 wait = $T_A+max(T_B-T_A-N_A,0)$	D.ops = N_A+N_B Q.ops = 0 wait = $T_A+max(T_B-T_A-N_A,0)$	-
(G)	D.ops = N_A+N_B Q.ops = 2 *(serialized $A \rightarrow B$)*	D.ops = N_A+N_B Q.ops = 0 wait = if $(T_A > T_B)$ T_A else $T_A+max(T_B-T_A-N_A,0)$ *(serialized $A \rightarrow B$)*	D.ops = N_A+N_B Q.ops = 0 wait = if $(T_A > T_B)$ $T_B+max(T_A-T_B-N_B,0)$ else $T_A+max(T_B-T_A-N_A,0)$	if $(T_A > T_B)$ $min(N_B, T_A-T_B)$ wait filled
(H)	D.ops = N_A+N_B Q.ops = 2 *(serialized $A \rightarrow B$)*	D.ops = N_A+N_B Q.ops = 0 wait = if $(T_A > T_B)$ T_A else $T_A+max(T_B-T_A-N_B,0)$ *(serialized $A \rightarrow B$)*	D.ops = N_A+N_B Q.ops = 0 wait = if $(T_A > T_B)$ $T_B+max(T_A-T_B-N_A,0)$ else $T_A+max(T_B-T_A-N_B,0)$	if $(T_A > T_B)$ $min(N_A, T_A-T_B)$ wait filled

D.ops = data ops, C.ops = control ops, Q.ops = queue ops, F.ops = predicated false ops, wait = serialization penalty
autonomous = internal activities of a PE, queue = PE responding to external events, T_i = time of channel availability

Table 3: Dynamic instruction cost of common spatial programming control idioms.

4. EVALUATION: CONTROL IDIOMS

In this section we evaluate the quantitative benefit of triggered instructions by examining a number of control idioms that arise frequently in spatially-programmed workloads. Table 3 compares implementations of each idiom on a triggered architecture to implementations on the PC+RegQueue and PC+Augmented architectures described in Section 2.

From this table we see some general patterns emerging. First, TI is never less efficient than a PC-based approach, i.e. it never requires more instructions. Second, TI removes all control operations such as branches. In a classic PC-based setting, the accepted rule of thumb is that about 1 in every 4-5 instructions is a branch [10]. In this setting TI's expected benefit would be around 20%. However in Section 5.3 we demonstrate that the fine-grained producer-consumer nature of spatially-programmed codes means that control makes up 44% of all operations, which increases the benefit of TI significantly.

Finally, TI removes the over-serialization problem presented in Section 2.3. This has several benefits, but they are harder to quantify directly. First, as the equations in Table 3 rows G-I demonstrate, there are certainly scenarios where over-serialization results in no penalty because the data arrives in the order that matches the static sequence chosen by the compiler. If the compiler can precisely schedule cross-PE data delivery rates then it is possible that this deficiency will never be exposed. In practice, the numerous sources of variable dynamic latency (memory hierarchy, network contention, data-dependent divergence, etc.) mean that there is plenty of opportunity to take advantage of the ability to break over-serialization.

Breaking over-serialization can be accomplished by finding independent operations. These can be found from two sources. The first source is local parallelism in the PE's dataflow graph, in which case computation can start as the data arrives, i.e., classical dynamic instruction reordering. The second source arises when the spatial compiler chooses to place unrelated sections of the overall algorithm dataflow graph onto a single PE, statically partitioning the registers between them and statically interleaving operations, i.e. compiler-directed multithreading. On a PC-based architecture, the serialization restriction is a significant barrier to a compiler's ability to statically partition one thread of control between unrelated sections of a single algorithm. The dynamic data production/consumption rates must be known to schedule the code—both for efficiency, and to avoid deadlock. On a TI architecture we expect compiler-directed multithreading of non rate-limiting PEs to be a common and important optimization.

To reiterate these benefits, since a TI architecture does not impose any ordering between instructions unless explicitly specified, it can gain the ILP benefits of an out-of-order issue processor *without the expensive instruction window and reorder buffer*. Simultaneously, a TI machine can take advantage of multi-threading *without duplicating data and control state*, but by the compiler partitioning resources as it sees fit. Of course there is a hardware cost associated with this benefit—the TI PE must have a scheduler (see Figure 7) that can efficiently evaluate the program's triggers.

5. EVALUATION: WORKLOADS

5.1 Approach

The objective of our quantitative evaluation in this section is twofold:

1. To demonstrate the effectiveness of a TIA-based spatial architecture compared to a traditional high-performance sequential architecture.

2. To demonstrate the benefits of using TIA-based PEs in a spatial architecture compared to PC-based PEs using the PC+RegQueue and PC+Augmented architectures described in Section 2.

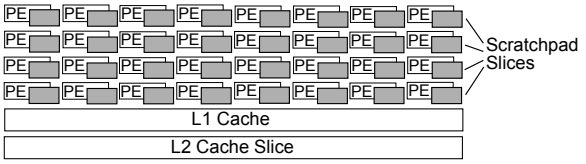
The main challenge with the first objective is that raw performance of a spatial accelerator is a function of area and memory bandwidth allocated to the accelerator, and parallelism available in the workload. Because spatial workloads generally exhibit good scalability, providing raw performance requires assessing a particular design point with a specific set of area/bandwidth values. However, since the purpose of this paper is to present a control paradigm for spatial architectures in general, we instead present performance numbers area-normalized against a typical host processor – namely a single 3.4 GHz out-of-order superscalar Intel® Core™ i7-2600 core.

Our evaluation fabric is a scalable spatial architecture built from an array of TIA PEs organized into *blocks*, which form the granularity of replication of the fabric. Each block contains a grid of interconnected PEs, a set of scratchpad slices distributed across the block, a private L1 cache, and a slice of a shared L2 cache that scales with the number of blocks on the fabric. Figure 8 provides an illustration of a block and the parameters that we use in our evaluation. Note that our evaluation PEs use 32-bit integer/fixed-point datapaths and do not include hardware floating point units (which is orthogonal to triggered instructions and beyond the scope of this evaluation). Area estimates of each PE were obtained via implementation feasibility analysis discussed further in Section 5.4. Area estimates for the caches, register files, multipliers, and on-chip network were added using existing industry results. As a reference, 12 blocks (each including PEs, caches, etc.) are about the same size as our baseline i7-2600 core (including L1 and L2 caches), normalized to the same technology node.

We developed a detailed cycle-accurate performance model of our spatial accelerator using Asim, an established performance modeling infrastructure [9]. We model the detailed microarchitecture of each TIA PE in the array, the mesh interconnection network, L1 and L2 caches, and DRAM.

We evaluate our spatial fabric on application kernels from a variety of domains. We do this under the assumption that the computationally-intensive portions of the workload will be offloaded from the main processor, which will handle peripheral tasks like setting up the memory and handling rare-but-slow cases. As a baseline we used sequential software implementations running on the i7-2600 host processor. When possible, we chose existing optimized workload implementations. In other cases, we auto-vectorized the workload using the Intel® C/C++ compiler (`icc`) version 13.0, enabling processor-specific ISA extensions.

For our second evaluation objective, we analyze how much of the overall speedup benefit is attributable to triggered instructions (as opposed to spatial programming in general) using the same framework described above. We demonstrate this by examining the critical loops that form the rate-limiting steps in the spatial pipeline of our workloads. We implemented the loops on spatial accelerators using the traditional program-counter based approaches. This analysis demonstrates how frequently the triggered instruction control idiom advantage presented in Table 3 translates to practical improvements.



PEs	32
Network	Mesh (1 cycle link latency)
Scratchpad	8KB (distributed)
L1 Cache	4KB (4 banks, 1KB/bank)
L2 Cache	24 KB shared slice
DRAM	200 cycle latency
Estimated Clock Rate	2 GHz

Figure 8: Block Illustration and Parameters.

5.2 Evaluation Application Kernels

For our analysis we have purposely chosen workloads spanning the space of data parallelism, pipeline parallelism, and graph parallelism. Table 4 presents an overview of the chosen kernels.

The triggered instruction versions of these kernels we implemented directly in our PE's assembly language and hand-mapped spatially across our fabric. (In the future we expect this to be done by automated tools from higher-level source code.) We offer these insights on the workloads' amenability to spatial programming:

- **AES-CBC:** Encryption with cipher-block chaining implemented using a memoized table in which byte substitution is performed. The algorithm is performed on a 4x4 grid of 8 bits apiece. One PE is responsible for providing the computation for a single byte, exposing 16-way parallelism.
- **Dense Matrix Multiply:** We adapt the SUMMA algorithm [11] by blocking problem size to the fabric. Input data is pipelined through loader PEs. Each worker PE operates on an 8*8 resultant matrix.
- **KMP String Search:** We adapt the Knuth-Morris-Pratt (KMP) [15] string search algorithm by slicing the text string into small segments and distributing it across a large number of PE workers. Another set of PEs are configured as pattern state machine generators and servers. A spatial implementation is able to slide the string window by simply rotating the logical order of the workers, discarding the block of text from the oldest worker and shifting in a new block in its place.
- **FFT:** We adapt a Fast Fourier Transform by blocking the complex-multiply butterfly structure into a size specific to our number of PEs. A control FSM re-uses this block to compose an FFT of arbitrary size.
- **Flow Classifier:** Network packet masking is parallelized by allocating different segments of the packet to different PEs. The hash key calculation is pipelined through a large number of PEs. The final comparison for matching flows is parallelized by processing multiple segments of the flow in parallel on multiple PEs.
- **Graph500-BFS:** The graph500 benchmark is meant to span multiple nodes of a supercomputer. We simulate what a single node would look like if enhanced with a spatial accelerator. We are able to pipeline the loading, testing, and updating of the nodes to expose a large number of in-flight memory requests.

Workload	Berkeley Dwarf [2]	Domain	Comparison Software Implementations
AES-CBC	Combinational Logic	Cryptography	Intel reference using AES - ISA extensions
KMP String Search	Finite State Machines	Various	Non-public optimized implementation
Dense Matrix Multiply	Dense Linear Algebra	Scientific Computing	Intel® MKL library implementation [11]
FFT	Spectral Methods	Signal Processing	FFT-W with auto-vectorization
Graph500-BFS	Graph Traversal	Supercomputing	Non-public optimized implementation
k-means Clustering	Dense Linear Algebra	Data mining	MineBench implementation with auto-vectorization
Merge Sort	Map/Reduce	Databases	Non-public optimized implementation
Flow classifier	Finite State Machines	Networking	Non-public optimized implementation
SHA-256	Combinational Logic	Cryptography	Intel reference (x86 assembly)

Table 4: Target Workloads for Evaluation.

- **k-means Clustering:** Our implementation maps the Euclidean distance function for a single cluster to a PE. Input data, along with the current nearest cluster, is streamed through the PEs in order to compare against all clusters.
- **Merge Sort:** Described previously in Section 2.2.
- **SHA-256:** The tight inner-loop is spatially mapped across PEs, with each function being mapped to a separate PE. Key generation is parallelized.

5.3 Performance Results

Figure 9 demonstrates the magnitude of performance improvement that can be achieved from using a spatially-programmed accelerator. Across our workloads, we observe area-normalized speedup ratios ranging from 3× (FFT) to around 22× (SHA-256) compared to the performance of the traditional core, with a geometric mean of 8×.

Now let us analyze how much of this benefit is attributable to the use of triggered instructions by comparing the rate-limiting inner loops of our workloads to implementations on spatial architectures using the PC+RegQueue and PC+Augmented control schemes.

Table 5 shows the average frequency of branches in the dynamic instruction stream for the PC-based spatial architectures. The branch frequency ranges from 8% to 70%, with an average of 50%. These inner loops are all very branchy and dynamic—far more than traditional sequential code.

This dynamism manifests itself as additional control cycles for both PC-based architectures, as shown in Figure 10. This figure shows the dynamic execution cycles for all architectures broken down into cycles spent on operations in each category defined in in Section 4. The cycle counts are all normalized to the number of D.ops (Data Computation operations) executed by PC+RegQueue. We augment this data with Figures 11 and 12, which respectively show the static and dynamic (average) instruction/op counts in the inner loops of rate-limiting steps for each workload. The data in these figures demonstrates that the control idiom efficiencies presented in Table 3 are applicable to real-world kernels. Specifically:

- TIA demonstrates a significant reduction in dynamic instructions executed compared to both PC+RegQueue (64%) and PC+Augmented (28%) on average, and an average performance improvement of 2.0× vs. PC+RegQueue and 1.3× vs. PC+Augmented in the critical loops. A large part of the performance gained by PC+Augmented over PC+RegQueue is from the reduction of Queue Management ops. TIA benefits from this too but gets a further performance boost over PC+Augmented from a reduction in Control ops and Predicated-False ops.
- An additional benefit of TIA over PC+Augmented comes from a reduction in Wait cycles. This is most evident in the k-means (50%), Graph500 (100%) and SHA-256 (40%) workloads. This is due to the ability of triggered instructions to avoid unnecessary serialization. Note that because these are critical rate-limiting loops in the spatial pipeline, there are fewer opportunities for multiplexing unrelated work onto shared PEs. Despite this, the workloads show benefits from avoiding over-serialization.
- The workload that sees the largest benefit from triggered instructions is Merge Sort. Merge Sort has the highest dynamic branch rate (70%) of all workloads on the PC+RegQueue architecture. It also spends a number of cycles polling queues. PC+Augmented eliminates all the queue-polling cycles, resulting in 1.6× performance improvement in the rate-limiting step. TIA further cuts down a large number of control cycles, leading to a further 2.3× performance improvement vs. PC+Augmented and a cumulative 3.7× performance benefit over PC+RegQueue.
- On the average, PC+Augmented does not see a significant benefit from predicated execution for these spatially-programmed workloads.
- Triggered instructions use a substantially smaller static instruction footprint. The reduction in footprint compared to PC+RegQueue is particularly significant — 62% on average. PC+Augmented's enhancements help reduce footprint but TIA still has 30% fewer static instructions on average.

The static code footprint of these rate-limiting inner loops is in general fairly small across all architectures. This observation, along with the real-world performance benefits we observed versus traditional high-performance architectures, provides strong evidence of the viability and effectiveness of the spatial programming model with small, tight loops arranged in a pipelined graph.

5.4 Implementation Feasibility Analysis

We collaborated with circuit-design experts to lay out a TIA PE in a state-of-the-art industry technology process. The resulting 2-stage pipelined PE has a comparable number of gate levels in the critical path to a high-performance commercial microprocessor. The large degree of replication in a spatial fabric would, however, justify even further design effort to optimize the PEs.

The hardware scheduler is the centerpiece of a TIA PE. Scheduler implementation cost is one of the primary factors that bounds the scalability of PE size in a triggered control model. Fortunately, the nature of spatial programming is such that small, efficient PEs are effective.

	AES	DMM	FFT	Flow Classifier	Graph-500	k-means	KMP Search	Merge Sort	SHA-256	Average
PC+RegQ	58%	50%	36%	50%	50%	69%	8%	70%	63%	50%
PC+Aug	6%	33%	11%	50%	40%	29%	14%	50%	22%	28%

Table 5: Percentage of dynamic instructions that are branches in rate-limiting step inner loop.

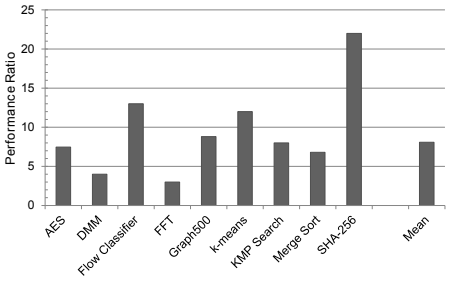


Figure 9: Area-normalized performance ratio of a TIA-based spatial accelerator compared to a high-performance out-of-order core.

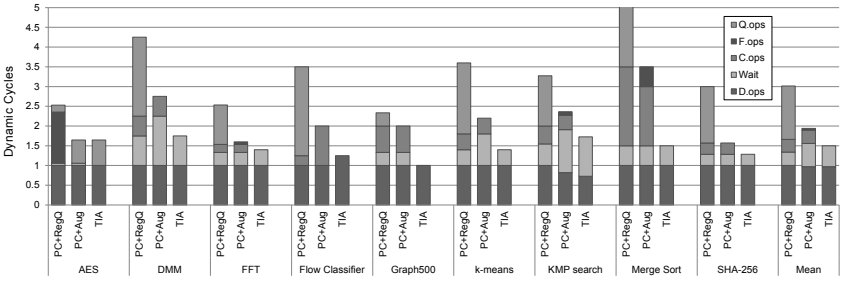


Figure 10: Breakdown of dynamic execution cycles in rate-limiting inner loops normalized to D.ops executed by PC+RegQueue.

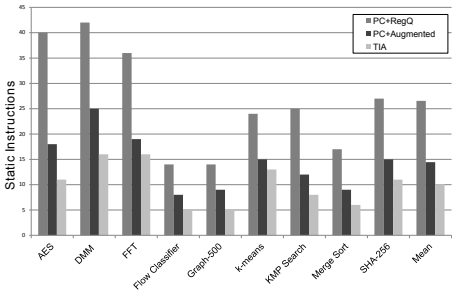


Figure 11: Static instruction counts for rate-limiting inner loops.

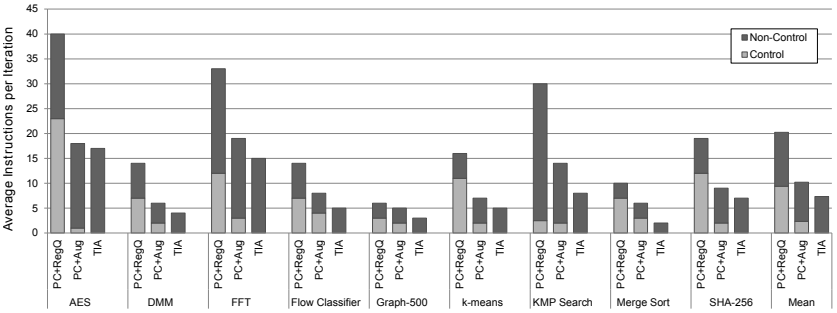


Figure 12: Average dynamic instruction counts for rate-limiting inner loops.

Our implementation analysis shows that the area cost of the TIA hardware scheduler is *less than 2%* of a PE's overall area, much of which is occupied by its architectural state (registers, input/output channel buffers, predicates and instruction storage), datapath logic (operand multiplexers, functional units, etc.) and microarchitectural control overheads—none of which are unique to triggered control. This is not surprising—the core of the TIA scheduler is essentially a few 1-bit wide trees of AND gates feeding into a priority encoder. For our chosen parameterization, this logic is dwarfed by everything else in the PE.

Similarly, scheduler power consumption is small compared to the rest of the PE. The scheduler logic does not consume dynamic power unless there is a change in predicate states. When this happens, the only wires that swing are the ones that are recomputing the changed control signals. This manner of computing control is more power-efficient than executing datapath instructions to compute the same results. In a degenerate scenario where the PE is walking down a sequence of stages in a gray-coded FSM, at most 1-2 predicate bits swing each cycle. The power consumed in this scenario is negligible.

6. RELATED WORK

We separate prior research into two categories: architectures with *autonomous* control flow where each PE internally controls itself, and *non-autonomous* control flow where each PE is controlled externally through outside control logic.

6.1 Autonomous PEs

Classic dataflow architectures such as [7, 1] trigger instructions when tokens associated with input data is ready. Multiple pipeline stages are devoted to marshalling tokens, distributing tokens, and scoreboarding which instructions are ready. A "Wait-Match" pipeline stage must dynamically pair incoming tokens of dual-input instructions. In contrast, TI expresses dependencies via single-bit predicate registers that are explicitly managed by the program, improving scheduler scalability and removing the token-related pipeline stages.

The RAW project is a coarse-grained computation fabric, consisting of 16 large cores with instruction and data caches that are directly connected through a register-mapped and circuit-switched network [24]. While applications written for RAW are spatially mapped, program counter management and serial execution of instructions reduces efficiency, and makes the cores on RAW sensitive to variable latencies, which TIA overcomes using instruction triggers.

The Asynchronous Array of simple Processors (AsAP) is a 36-PE processor for DSP applications, with each PE executing independently using instructions in a small instruction buffer and communicating using register-mapped network ports [27]. While early research on AsAP avoided the need to poll for ready data, later work extended the original architecture to support 167-PEs and zero-overhead looping to reduce control instructions [25]. Triggered instructions not only reduce the amount of control instructions but also enable data-driven instruction issue, overcoming the serialization of AsAP's program-counter based PE.

Picochip is a commercially available 308-PE accelerator for DSP applications [21]. Each PE has a small instruction and data buffer, and communication is performed with explicit *put* and *get* commands. A strength of Picochip is compute density, but the architecture is limited to serial 3-way LIW instruction issue using a program counter. Triggered instructions enable control flow at low cost and dynamic instruction issue dependent on data arrival, resulting in less instruction overhead.

6.2 Non-autonomous PEs

Transport-Triggered Architectures [14] is a scheme where the functional units in the system are exposed to the compiler, which then uses MOV operations to explicitly route data through the transport network. Overall control flow is maintained by a global program counter. Operation execution is triggered by the arrival of data from the network, but no other localized control exists.

TRIPs is an *explicit dataflow graph execution* (EDGE) processor which utilizes many small PEs to execute general-purpose applications [4]. TRIPs dynamically fetches and schedules large VLIW instruction blocks across the small PEs using centralized program-counter based control tiles. While large reservation stations within each PE enable "when-ready" execution of instructions, only single-bit predication is used within PEs to manage small amounts of control flow.

WaveScalar is a dataflow processor for general-purpose applications that does not utilize a program counter [23]. A PE consists of an ALU, input and output network connections, and a small window of 8 instructions. Blocks of instructions known as waves are mapped down onto the PEs, and additional "WaveAdvance" instructions are allocated at the edges to help manage coarse grained or loop-level control. Conditionals are handled by converting control flow instructions to data flow, resulting in filtering instructions that conditionally pass values to the next part of the dataflow graph. In WaveScalar there is no local PE register state; when an instruction issues the result must be communicated to another PE across the network.

DySER integrates a circuit-switched network of ALUs inside the datapath of contemporary processor pipeline [12]. DySER maps a single instruction to each ALU and does not allow memory or complex control flow operations within the ALUs. TIA enables efficient control flow and spatial program mapping across PEs, enabling high-utilization of ALUs with PEs without the need for an explicit control core. Other recent work such as Garp [13], Chimaera [26], and ADRES [18] similarly integrate LUT-based or coarse grained reconfigurable logic controlled by a host processor, either as a coprocessor or within the processor's datapath.

MATRIX [20] is an array of 8-bit function units with a configurable network. With different configurations, MATRIX can support VLIW, SIMD or Multiple-SIMD computations. The key feature of the MATRIX architecture was claimed to be its ability to deploy resources for control based on application regularity, throughput requirements and space available.

PipeRench [22] is a coarse-grained RL system designed for virtualization of hardware to support high-performance custom computations through self-managed dynamic reconfiguration. It is constructed from 8-bit Processing Elements. The functional unit in each PE contains eight 3-input LUTs that are identically configured.

In contrast to all these non-autonomous control based approaches, TIA enables complex fine-grained control at each PE, which makes it applicable to a broader domain of spatial workloads.

7. CONCLUSION

We believe that spatial parallelism is a promising computing paradigm with the potential to achieve significant performance improvement over traditional high-performance architectures for a number of important workloads, many of which do not exhibit uniform data parallelism. Our simulated performance estimates on a triggered-instruction based spatial architecture confirm the potential of this style of computing, showing an average area-normalized performance that is 8× better than a high-end sequential processor across a range of workloads.

Triggered instructions provide a uniform solution to the control problem for a PE in a spatially-programmed architecture, allowing the PE to execute autonomous control loops efficiently as well as react quickly to messages on communication channels. The same mechanism also avoids over-serialization, providing the benefits of dynamic instruction reordering and multithreading without any additional hardware. Our evaluation demonstrates the cumulative benefits of all these effects, with our triggered-instruction PE achieving 2.0× better performance than a baseline PE with PC-based control, and 1.3× better performance than an optimized version.

The triggered control model is feasible within a spatially-programmed environment because the amount of static instruction state that must be maintained in each PE is small, allowing for inexpensive implementation of a triggered-instruction hardware scheduler. Our implementation analysis confirms this, showing that the scheduler occupies less than 2% of the area of the PE.

These results provide a solid foundation of evidence for the merit of a triggered-instruction based spatial architecture. The ultimate success of this paradigm will be premised on overcoming a number of challenges, including providing a tractable memory model, dealing with the finite size of the spatial array, and providing a high-level programming and debugging environment. Our ongoing work makes us optimistic that these challenges are surmountable.

Acknowledgments

We thank James Brodman, Rakesh Komuravelli, Neal Oliver, Shaojuan Zhu, Hisham Qayum and Jerome David Rosen for their valuable contributions to workload analysis. We benefited greatly from the circuit implementation analysis provided by Jagdish Patil, George Clark and Jonathan Enoch. We acknowledge Tao Wang and Peng Li for contributions to earlier incarnations of these ideas. Finally, we greatly appreciate the many insightful discussions with Arvind, Kermin Elliott Fleming and Arch Robison.

8. REFERENCES

[1] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L.

Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[3] Bluespec, Inc. Bluespec System Verilog Reference Guide. 2007.

[4] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37(7):44–55, July 2004.

[5] K. M. Chandy and J. Misra. *Parallel Program Design: a Foundation*. Addison-Wesley, 1988.

[6] K. Compton and S. Hauck. Reconfigurable Computing: A Survey Of Systems and Software. *ACM Computer Survey*, 34(2):171–210, June 2002.

[7] J. B. Dennis and D. P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *Proceedings of the 2nd annual Symposium on Computer Architecture*, pages 126–132, 1975.

[8] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.

[9] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A Performance Model Framework. *Computer*, 35(2):68–76, 2002.

[10] J. S. Emer and D. W. Clark. A Characterization of Processor Performance in the vax-11/780. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA)*, pages 301–310, 1984.

[11] R. A. V. D. Geijin and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical report, 1997.

[12] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of 17th International Conference on High Performance Computer Architecture (HPCA)*, 2011.

[13] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.

[14] J. Hoogerbrugge and H. Corporaal. Transport-Triggering vs. Operation-Triggering. In *Lecture Notes in Computer Science 786, Compiler Construction*, pages 435–449. Springer-Verlag, 1994.

[15] D. E. Knuth, J. Morris, and V. R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[16] H. T. Kung. The CMU Warp Processor. In F. A. Matsen and T. Tajima, editors, *Supercomputers: Algorithms, Architectures, and Scientific Computation*, pages 235–247. 1986.

[17] A. Marquardt, V. Betz, and J. Rose. Speed and Area Tradeoffs in Cluster-Based FPGA Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):84 –93, Feb. 2000.

[18] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proceedings of 13th International Conference on Field-Programmable Logic and Applications*, pages 61–70, Sep. 2003.

[19] D. G. Merrill and A. S. Grimshaw. Revisiting Sorting for GPGPU Stream Architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 545–546, 2010.

[20] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, Apr. 1996.

[21] G. Panesar, D. Towner, A. Duller, A. Gray, and W. Robbins. Deterministic Parallel Processing. *International Journal of Parallel Programming*, 34(4):323–341, Aug. 2006.

[22] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Taylor. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Proceedings of the 2002 IEEE Custom Integrated Circuits Conference*, pages 63–66, May 2002.

[23] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The WaveScalar Architecture. *ACM Transactions on Computer Systems*, 25(2):4:1–4:54, May 2007.

[24] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[25] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas. A 167-Processor Computational Platform in 65 nm CMOS. *IEEE Journal of Solid-State Circuits*, 44(4):1130–1144, April 2009.

[26] Z.-A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 225–235, Jun. 2000.

[27] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. An Asynchronous Array of Simple Processors for DSP Applications. In *Solid-State Circuits Conference (ISSCC), Digest of Technical Papers*, pages 1696–1705, Feb. 2006.